EAST WEST UNIVERSITY

Department of Computer Science and Engineering

Course Title: Algorithms

Course Code: CSE 246

Semester: Spring 2023

Section : 02

Submitted To

Redwan Ahmed Rizvee

Lecturer

Department of Computer Science and Engineering

Submitted by :

Name: Md. Rakibur Rahman

Student ID: 2019-3-60-113

Lab : 01 –

#Problem 1: Upper and Lower Bound

Description:
Given a sorted array A of size N and a search value S, you have to find the upper and lower bound value for S.
In the first line, you will be given N and S. In the second line, you will be given N values denoting the sorted array.
You have to print the upper and lower bound respectively with a single blank space. Keep a new line character after each output. Print the answer using 0 based indexing.

Upper Bound: The index of the smallest value in the sorted array which is greater than S. For repeating such values consider the largest or the right-most index. If the largest value in the array is smaller than S. Then the upper bound is the size of the array.

Lower Bound: The index of the largest value in the sorted array which is smaller or equal to S. For repeating such values consider the smallest or the left-most index. If S is smaller than the smallest value in the array, consider the lower bound as 0.

Limits:
1<=|A|<=100000

Code :

```
#include <bits/stdc++.h>

using namespace std;

int lower_bound(int a[], int s, int n)

{

if (s < a[0])

return 0;

int idx, l = 0, r = n - 1;

while (l <= r)

{

int m = l + (r - l) / 2;

if (a[m] <= s)

{

idx = m;
```

```
l = m + 1;
}
else
r = m - 1;
}
int fidx, val = a[idx];
l = 0, r = n - 1;
while (l <= r)
{
int m = l + (r - l) / 2;
if (a[m] >= val)
{
fidx = m;
r = m - 1;
}
else
l = m + 1;
}
return fidx;
}
int upper_bound(int a[], int s, int n)
{
if (a[n - 1] < s)
return n;
int idx, l = 0, r = n - 1;
```

```
while (l <= r)
{
int m = l + (r - l) / 2;
if (a[m] > s)
{
idx = m;
r = m - 1;
}
else
l = m + 1;
}
return fidx;
}
int upper_bound(int a[], int s, int n)
{
if (a[n - 1] < s)
return n;
int idx, l = 0, r = n - 1;
while (l <= r)
{
int m = l + (r - l) / 2;
if (a[m] > s)
{
idx = m;
r = m - 1;
```

```
}
else
l = m + 1;
}
int fidx, val = a[idx];
l = 0, r = n - 1;
while (l <= r)
{
int m = l + (r - l) / 2;
if (a[m] <= val)
{
fidx = m;
l = m + 1;
}
else
r = m - 1;
}
return fidx;
}
int main()
{
int n, s;
cin >> n >> s;
int a[n];
for (int i = 0; i < n; i++)
```

```
cin >> a[i];

int lb = lower_bound(a, s, n);

int ub = upper_bound(a, s, n);

cout << ub << " " << lb << "\n";

}
```

#Problem 2: Finding Square Root

Description:
You will be given an integer number. You need to calculate the square root of that number up to three decimal places. So, your precision should match three places after the decimal. You can not use the built-in function `sqrt` here.

Code :

```
#include <iostream>

#include <iomanip>

using namespace std;


int main()

{

  int num;

  cin >> num;

  float temp, sqrt;


  sqrt = num / 2;

  temp = 0;
```

```
  while (temp != sqrt)

  {

    temp = sqrt;

    sqrt = (num / temp + temp) / 2;

  }



  cout << fixed << setprecision(3) << sqrt << endl;

  return 0;

}
```

Lab : 02 -

#Problem 1: Height of the students

Description:
You will be dealing with the heights of the students here. First, you will be given an integer value N denoting the number of students in the class. Then you will be given N numbers (can be floating point numbers) denoting the *height* of each student in some unit. The i^{th} number will denote the *height* of the i^{th} student. After that you will be given a value K.

In the assembly students are arranged from the smallest to largest height. You have to print the roll which will be standing in the K^{th} position in such an arrangement. The rolls follow 1 based indexing. In the case of repeating heights, you need to print the student with a lesser roll.

Limits:
1<=N<=100000
1<=k<=100000
100 <= height <= 5000

Code :

```
#include<bits/stdc++.h>

using namespace std;



int main()

{

   int m, p;
```

```
    cin >> m;
    vector < pair< double, int >> students(m);
    for (int j = 0; j < m; j = j+1) {
        cin >> students[j].first;
        students[j].second = j+1;
    }
    cin >> p;


    sort(students.begin(), students.end());


    cout << students[p-1].second << endl;


    return 0;
}
```

#Problem 2: Min Distance between the points of 1D coordinates

Description:
You will be given N, 1D co-ordinates lying in the number line. You have to print the absolute minimum distance found between two given coordinates.

Limits:
1<=N<=100000

Code :

```
#include<bits/stdc++.h>


using namespace std;


int main() {
```

```
    int m;
    cin >> m;

    double  coords[m];
    for (int j = 0; j < m; j = j+1) {
        cin >> coords[j];
    }

    sort(coords, coords + m);

    int minDist = abs(coords[0] - coords[1]);
    for (int j = 1; j < m - 1; j= j+1) {
        int dist = abs(coords[j] - coords[j + 1]);
        if (dist < minDist) {
            minDist = dist;
        }
    }

    cout << minDist << endl;

    return 0;
}
```

#Problem 3: Finding Majority Element - 1

Description:
In this problem, you will be given N numbers. You need to find the element which occurred maximum number of times. If there are ties in frequencies, return the element which is comparatively greater.

In the first line, you will be given an integer N. In the following line, you will be given N integer values A_i (1<=i<=N).
For each input there will be a single line of output. Printing the element with the maximum number of occurrences. For ties in occurrences, print the element with the maximum value.

Limits:
1<=N<=100000, -10^5 <= A_i <= 10^5

Code :

```
#include <bits/stdc++.h>

using namespace std;


int main() {

   int m;

   cin >> m;


   int a[m];

   for (int i = 0; i < m; i++) {

      cin >> a[i];

   }


   sort(a, a + m);


   int maxCount = 1, currCount = 1;

   int maxElement = a[0], currElement = a[0];


   for (int i = 1; i < m; i++) {

      if (a[i] == currElement) {
```

```
            currCount++;
        } else {
            if (currCount > maxCount || (currCount == maxCount && currElement > maxElement)) {
                maxCount = currCount;
                maxElement = currElement;
            }
            currElement = a[i];
            currCount = 1;
        }
    }

    if (currCount > maxCount || (currCount == maxCount && currElement > maxElement)) {
        maxCount = currCount;
        maxElement = currElement;
    }

    cout << maxElement << endl;

    return 0;
}
```

Lab -03 :

#Problem 1: Simple Coin Change

Description:
You will be given N coins and an amount K. You can use each coin an infinite number of times. You have to calculate the minimum number of coins needed to make the amount K.

In the first line you will be given N and K. In the following line, you will be given N values $C_1,C_2,C_3,...,C_N$ denoting N coins.

For each set of input, there will be a single line of output denoting the minimum number of coins to make K.

Limits
1<=N<=100000
1<=k<=1000000
Each coin C_i will have values between 1 and 10000 inclusive

Code :

```
#include <iostream>

using namespace std;



int main() {

   int p, m;

   cin >> p >> m;



   int coins[p];

   for (int i = 0; i < p; i= i+1)

      cin >> coins[i];



   int minCoins[m+1];

   for (int i = 0; i <= m; i = i+1)

      minCoins[i] = m+1;

   minCoins[0] = 0;



   for (int i = 1; i <= m; i= i+1)

      for (int j = 0; j < p; j = j+1)

         if (coins[j] <= i && minCoins[i-coins[j]] + 1 < minCoins[i])
```

```
            minCoins[i] = minCoins[i-coins[j]] + 1;


   if (minCoins[m] > m)

      cout << "-1\p";

   else

      cout << minCoins[m] << endl;


   return 0;

}
```

#Problem 2: Fractional Knapsack

Description:
You will be given N items and a knapsack weight W. Each item N[i] has two elements, its positive benefit per unit b[i] and its total weight w[i]. You have to choose the elements in such a way that you can maximize your total benefit. You can take a fractional amount of weights for each item.

In the first line, you will be given N and W. In the first following line you will be given N values, each element denotes the positive benefit per unit where i^{th} value is for the i^{th} element. In the second following line, you will again be given N values, each denoting the weights where i^{th} value denotes the total available weight for the i^{th} element.

For each set of input, there will be a single line of output denoting the optimal value that maximize your total benefit.

Limits
1<=N<=100000
1<=W<=100000
1<=b[i]<=100
1<=w[i]<=100000

Code :

```
#include <iostream>

#include <algorithm>


using namespace std;
```

```
struct Item {
    double b;
    int w;
};

bool cmp(Item a, Item b) {
    return a.b > b.b;
}

int main() {
    int n, W;
    cin >> n >> W;

    Item items[n];
    for (int i = 0; i < n; i++) {
        cin >> items[i].b;
    }
    for (int i = 0; i < n; i++) {
        cin >> items[i].w;
    }

    sort(items, items + n, cmp);
    int    totalBenefit = 0;
    int remainingWeight = W;
```

```
    for (int i = 0; i < n && remainingWeight > 0; i++) {
        int take = min(remainingWeight, items[i].w);
        totalBenefit += take * items[i].b;
        remainingWeight -= take;
    }

    cout.precision(2);
    cout << fixed << totalBenefit << endl;

    return 0;
}
```

#Problem 3: Activity Scheduling

Description:
You will be given N tasks, each having a starting and ending time. You can complete only one task at a time. You have to schedule the tasks in such an order so that the number of completed tasks maximizes.

In the first line, you will be given a value N. In the following N lines, you will be given the information of each N tasks where the i^{th} line contains the starting time S_i and ending time E_i for the i^{th} task.

For each set of input, there will be a single line of output denoting the number of maximum possible tasks that can be performed without violating conflicting constraint. Overlapping between consecutive tasks' ending and starting time is permissible, e.g., one task ended at 3 and another task started at 3, in this case, both tasks can be considered without considering the overlapping conflict.

Limits
$1<=N<=100000$
$1<=S_i<=1000$
$1<=E_i<=1000$ always $E_i>=S_i$ for all i

Code :

```
#include <iostream>
```

```
#include <algorithm>
using namespace std;

struct Task {
   int start, end;
};

bool cmp(const Task& a, const Task& b) {
   return a.end < b.end;
}

int main() {
   int n;
   cin >> n;
   Task tasks[n];
   for (int i = 0; i < n; i = i+1) {
      cin >> tasks[i].start >> tasks[i].end;
   }
   sort(tasks, tasks + n, cmp);
   int ans = 1, last_end = tasks[0].end;
   for (int i = 1; i < n; i = i+1) {
      if (tasks[i].start >= last_end) {
         ans = ans+1;
         last_end = tasks[i].end;
      }
```

```
    }

    cout << ans << endl;

    return 0;

}
```

Lab -04 :

#Problem 1: Basic Prime Checking

Description:
In this problem, first you will be given the number of test cases T. Then you will be given T numbers. For each number t_i ($1<=i<=T$), you need to find if it's a prime number or not. Look at the input output section for more clarification.

Limits
$1<=T<=100$
$1<=t_i<=10^9$
= No solve .

#Problem 2: Finding the Pattern's occurrences

Description:
In the first line, you will be given a text T and in the following line you will be given a pattern P. You need to find the occurrences of P in T. Print each occurrence (start and end index in T) in separate lines keeping the ascending order of the starting indexes. Look at the input output section for more clarification. The output will follow 0 based indexing.

Limits
P and T both will always contain only English upper case letters, 'A' <= P_i, T_i <='Z'.
$1<=|P|<=1000$
$1<=|T|<=10000000$

Code :

```
#include<stdio.h>

#include<string.h>



int main()

{

    int i, j, k;
```

```
    char T[10001], P[1001];


    scanf("%s %s", &T, &P);
    int a=strlen(T);
    int b=strlen(P);


    for(i=0; i<=a-b; i = i+1)
    {
        int flg =0;


        for(j=i, k=0; j<i+b; j = j+1, k = k+1)
        {
            if(T[j]!=P[k])
            {
                flg=1;
                break;
            }
        }
        if(flg==0) printf("%d %d\n", i, i+b-1);
    }
    return 0;
}
```

#Problem 3: Calculate Prefix Function of a String

Description:
In this problem, you will be given a string P. You need to calculate the prefix function's value of P. For each length prefix P_i (1 length, 2 length, 3 length, etc.) of P, prefix function calculates the length of the maximum prefix that matches with the suffix of P_i .

Let the length of the given string be m. Then the output will have a single line containing m integer values separated with a single space denoting the length of the maximum prefix that matches with the suffix for each length prefix of P_i .

Limits
P will always contain only English upper case letters, ‘A’ <= P_i, T_i <=’Z’.
1<=|P|<=100000
= No solve .

Lab -05 :

#Problem 1: Complex Coin Change

Description:
You will be given N coins and an amount K. You can use each coin an infinite number of times. You have to print the minimum number of coins needed to make the amount K.

In the first line you will be given N and K. In the following line, you will be given N values denoting N coins.

Limits
1<=N<=30
1<=K<=10000

Code :

```
#include<bits/stdc++.h>

using namespace std;



int dp[10010];



int minCoins(int amount, vector<int> coins) {

   if(amount == 0)

      return amount;

   if(dp[amount] != -1)

      return dp[amount];

   int ans = INT_MAX;
```

```
    for(auto coin: coins)
        if(amount-coin >= 0)
            ans = min(ans+0LL, minCoins(amount-coin, coins)+1LL);
    return dp[amount] = ans;
}

int main() {
    int N, K;
    vector<int> coins;
    cin >> N >> K;
    while(N--) {
        int coin; cin >> coin;
        coins.push_back(coin);
    }
    memset(dp, -1, sizeof(dp));
    cout << minCoins(K, coins) << endl;

    return 0;
}
```

#Problem 2: Complex Coin Change with Coin Print

Description:
You will be given N coins and an amount K. You can use each coin an infinite number of times. You have to make the amount K using the minimum number of coins. You also have to print which coins have been taken with their usage count in ascending order over the coin's value aka smaller valued coin will be printed first.

In the first line, you will be given N and K. In the following line, you will be given N values denoting N coins.

For each input, print the coins with their usage count in ascending order. If there exists multiple solutions print any of them. Print each coin's output in separate lines. Look at the input output section for more clarification.

Limits
1<=N<=30
1<=K<=10000

Code :

```
#include<bits/stdc++.h>

using namespace std;



int main() {

   // this is the tabular method or bottom up approach

   int N, K;

   cin >> N >> K;

   int coins[N];

   for(int i = 0; i < N; i++) {

      cin >> coins[i];

   }

   sort(coins, coins+N);



   int a[N][K+1];



   int i = 0;

   int j = 0;

   a[i][j] = 0;

   for(j = 1; j < K+1; j++) {

      if(coins[i] > j)
```

```
        a[i][j] = 0;
    else if(j%coins[i] == 0)
        a[i][j] = j/coins[i];
    else
        a[i][j] = 0;
}


for(int i = 1; i < N; i++) {
    for(int j = 0; j < K+1; j++) {
        if(coins[i] > j) {
            a[i][j] = a[i-1][j];
        }
        else if(a[i-1][j] == 0) {
            a[i][j] = 1+ a[i][j-coins[i]];
        }
        else {
            int t = j-coins[i];
            if(t != 0 && a[i][t] == 0)
                a[i][j] = a[i-1][j];
            else
                a[i][j] = min(a[i-1][j], 1+a[i][t]);
        }
    }
}
```

```
    // find the coins
    map<int, int> m;
    i = N-1;
    j = K;
    while(true) {
        if(j == 0) {
            break;
        }
        if(a[i][j] != a[i-1][j]) {
            m[coins[i]]++;
            j = j-coins[i];
        } else {
            i--;
        }
    }

    for(auto data: m) {
        cout << data.first << " " << data.second << endl;
    }

    return 0;
}
```

#Problem 3: 0/1 Knapsack

Description:
You will be given N items and a knapsack weight W. Each item N[i] has two elements, its total positive benefit b[i] and its total weight w[i]. You have to choose the elements in such a way that

you can maximize your total benefit not exceeding W. You can not take a fractional amount of weight for an item.

In the first line, you will be given N and W. In the first following line you will be given N values, each element denotes the total positive benefit where i^{th} value is for the i^{th} element. In the second following line, you will again be given N values, each denoting the weights where i^{th} value denotes the total weight for i^{th} element.

For each input, you have to output the maximum positive benefit you can achieve not exceeding the knapsack weight.

Limits
1<=N<=30
1<=W<=1000

Code :

```
#include<bits/stdc++.h>

using namespace std;



long long dp[35][1005];

int wt[35], val[35];

long long knapsack(int idx, int wt_left) {

   if(wt_left == 0)

      return 0;

   if(idx < 0)

      return 0;

   if(dp[idx][wt_left] != -1)

      return dp[idx][wt_left];



   long long ans = knapsack(idx-1, wt_left);

   if(wt_left- wt[idx] >= 0)

      ans = max(ans, knapsack(idx-1, wt_left- wt[idx])+ val[idx]);
```

```
    return dp[idx][wt_left] = ans;
}

int main() {
    int N, W;
    cin >> N >> W;
    for(int i = 0; i < N; i++) {
        cin >> val[i];
    }
    for(int i = 0; i < N; i++) {
        cin >> wt[i];
    }

    memset(dp, -1, sizeof(dp));
    cout << knapsack(N-1, W) << endl;

    return 0;
}
```

#Problem 4: LCS and Path Print

Description:
Given two strings M and N. You need to print the longest common subsequence (LCS) length found between M and N. You also need to print a lcs which exists between M and N.

In the first line, you will be given M and in the second line you will be given N. M and N will contain only digits or English alphabets.

As output, in the first line print the lcs length. In the second line, print a lcs which exists between M and N. If there exists multiple solutions print any of them. Look at the input output section for more clarification.

Limits

1<=|M|<=40
1<=|N|<=40
|M|, |N| denote the length of string M and N respectively.

Code :

```
#include<bits/stdc++.h>

using namespace std;


int a1[41], a2[41];

int c[41][41];


class dir {

   public:

   int i, j;

};


dir d[41][41];


int main() {

   string M, N;

   cin >> M >> N;


   // this is tabular method or botton up approach


   for(int i = 0; i < M.size(); i++)

      a1[i+1] = M[i];

   for(int j = 0; j < N.size(); j++)
```

```
    a2[j+1] = N[j];


memset(c, 0, sizeof(c));


for(int i = 1; i <= M.size(); i++) {
    for(int j = 1; j <= N.size(); j++) {
        if(a1[i] == a2[j]) {
            c[i][j] = 1+ c[i-1][j-1];
            d[i][j].i = i-1;
            d[i][j].j = j-1;
        }
        else {
            if(c[i][j-1] >= c[i-1][j]) {
                c[i][j] = c[i][j-1];
                d[i][j].i = i;
                d[i][j].j = j-1;
            } else {
                c[i][j] = c[i-1][j];
                d[i][j].i = i-1;
                d[i][j].j = j;
            }
        }
    }
}
cout << c[M.size()][N.size()] << endl;
```

```
// path print
stack<char> st;
int m = M.size(), n = N.size();
while(true) {
   if(m == 0 || n == 0) {
      break;
   }
   if(d[m][n].i == m-1 && d[m][n].j == n-1) {
      st.push((int)a1[m]);
      m = m-1;
      n = n-1;
   }
   else if(d[m][n].i == m && d[m][n].j == n-1) {
      n = n-1;
   }
   else if(d[m][n].i == m-1 && d[m][n].j == n) {
      m = m-1;
   }
}

while(!st.empty()) {
   cout << st.top();
   st.pop();
}
```

```
    cout << endl;



    return 0;

}
```

Lab 6 :

#Problem 1: LIS, Longest Increasing Subsequence

Description:
You will be given an array of size N having N integer numbers. You need to calculate the longest increasing subsequence length of the array.

In the first line you will be given N. In the following line you will have N integer numbers.

Limits
1<=N<=30

Code :

```
#include<bits/stdc++.h>

using namespace std;



int dp[70];



// top down approach

// time complexity O(n2)

int lis(vector<int> &v, int i) {

    if(dp[i] != -1)

        return dp[i];

    int ans = 1;

    for(int j = 0; j < i; j++) {

        if(v[i] > v[j]) {
```

```
            ans = max(ans, lis(v, j)+ 1);
        }
    }
    return dp[i] = ans;
}

int main() {
    int N; cin >> N;
    vector<int> v(N);
    for(int i = 0; i < N; i++)
        cin >> v[i];

    memset(dp, -1, sizeof(dp));
    int ans = 0;
    for(int i = 0; i < N; i++) {
        ans = max(ans, lis(v, i));
    }
    cout << ans << endl;

    return 0;
}
```

#Problem 2: Hill Climbing

Description:
An avenger is trying to climb a very dangerous hill. There are some places on the hill which are very steep and dangerous to climb. Also, there are some places on the hill which are pretty flat and possess no difficulties.

This hill climbing problem can be modeled as a 2D grid problem. Where the grid has M rows and N columns. Analogically here, M denotes the height and N denotes the width of the hill respectively.

The climber is at the bottom now. He will start climbing the hill aka start traversing the grid from the bottom row. In the grid, for each cell a value is given which denotes the danger lying in that cell. If the climber comes in this cell, it will add danger for him in the path to climb the hill. In this problem, you need to find the optimal path for the climber to reach the top of the hill, **minimizing** the total danger of climbing. From each cell, a climber can make three possible moves which has been shown in the following figure,

**climbed complete hill*

Can go here	Can go here	Can go here	
	Current cell		

**hill bottom*

In the first line, you will be given two values M and N denoting the number of rows and columns of the grid. Then you will have M lines of values where each line will have N values. Here the values lying in the i^{th} line denote the values of the i^{th} row of the grid. In such i^{th} line, j^{th} value denotes the danger value of grid cell (i,j).

In the output, you need to print the minimum possible danger value for the climber to reach the top row. Outside of the bottom row means, he did not start climbing and outside of the top row means, he has finished climbing. The climber can not move right from any cells belonging to the rightmost column, similarly can not move left from the leftmost column, because it would bring death to him.

Limits
1<=M,N<=40
0<=Grid cell [i,j] <=1000 for all (i,j) pairs

Code :

```
#include<bits/stdc++.h>

using namespace std;



long long Hill[100][100];

long long dp[500][500];
```

```
// this is top down approach
// time complexity O(m*n)
long long Hillclimb(int i, int j, int m, int n) {
   if(i >= m)
      return 0;
   if(j <= -1 || j >= n)
      return INT_MAX;

   if(dp[i][j] != -1)
      return dp[i][j];

   long long ans = Hillclimb(i+1, j-1, m, n)+ Hill[i][j];
   ans = min(ans, Hillclimb(i+1, j, m, n)+ Hill[i][j]);
   ans = min(ans, Hillclimb(i+1, j+1, m, n)+ Hill[i][j]);

   return dp[i][j] = ans;
}

int main() {
   int M, N;
   cin >> M >> N;
   for(int i = 0; i < M; i++) {
      for(int j = 0; j < N; j++) {
         cin >> Hill [i][j];
```

```
        }
    }
    memset(dp, -1, sizeof(dp));
    long long ans = INT_MAX;
    for(int j = 0; j < N; j++) {
        ans = min(ans, Hillclimb(0, j, M, N));
    }
    cout << ans << endl;


    return 0;
}
```

#Problem 3: MCM Calculation

Description:
You will be given N number of 2D matrices' information aka row and column value for each matrix. You need to calculate the minimum number of operations needed to calculate the multiplication of all the matrices.

In the first line you will be given N. In the following N lines you will find the row and column information of each given matrix, ith line contains the information of ith matrix.

Code :

```
#include<bits/stdc++.h>

using namespace std;



int dp[50][50];



// this is top down approach

// time complexity O(n3)

int mcm(int i, int j, vector<int> &v) {
```

```
    if(i == j)
        return 0;
    if(dp[i][j] != -1)
        return dp[i][j];
    int mini = INT_MAX;
    for(int k = i; k < j; k++)
        mini = min(mini, (v[i-1]* v[k]* v[j])+ mcm(i, k, v)+ mcm(k+1, j, v));

    return dp[i][j] = mini;
}

int main() {
    int N; cin >> N;
    vector<int> v;
    for(int i = 0; i < N; i++) {
        int x, y;
        cin >> x >> y;
        if(i == 0) {
            v.push_back(x);
            v.push_back(y);
            continue;
        }
        v.push_back(y);
    }
```

```
    memset(dp, -1, sizeof(dp));

    cout << mcm(1, N, v) << endl;



    return 0;

}
```

Lab -07 :

#Problem 1: Unweighted Shortest Path - Undirected

Description: In this problem you will be given an undirected unweighted (or each edge equally weighted) graph G, a source vertex S and a destination vertex D. You need to calculate the shortest distance between S to D. In this problem, the shortest distance between two vertices means using the minimum number of edges to reach from one vertex to the other.

In the first line you will be given two integers V and E denoting the number of vertices and edges of this graph respectively. All the vertices will have ids between 1 to V. In the i^{th} line you will have two integers x (1<=x<=V) and y (1<=y<=V) which denotes there is an undirected edge between x and y. After E lines of input, you will get two integers S and D (1<=S<=V, 1<=D<=V) denoting the source and destination vertices of the problem.

As output, you will print an integer number d which will denote the minimum distance between S and D. If there lies no path between S and D print “INF” without quotes.

Limits
1<=V<=100000, 1<=|E|<=100000

Code :

```
#include<bits/stdc++.h>

using namespace std;

#define N 100005



vector<int> graph[N];

bool vis[N];

queue<int> q;

int level[N];
```

```
void bfs(int src, int des) {
    vis[src] = true;
    q.push(src);

    while(!q.empty()) {
        int node = q.front();
        q.pop();

        for(auto child: graph[node]) {
            if(vis[child]) {
                continue;
            }
            q.push(child);
            vis[child] = true;
            level[child] = level[node]+1;
        }
    }

    if(level[des] != 0)
        cout << level[des] << endl;
    else
        cout << "INF" << endl;
}
```

```
int main() {

   int V, E;

   cin >> V >> E;

   for(int i = 1; i <= E; i++) {

      int x, y;

      cin >> x >> y;

      graph[x].push_back(y);

      graph[y].push_back(x);

   }

   int S, D;

   cin >> S >> D;

   bfs(S, D);



   return 0;

}
```

#Problem 2: Bicoloring

Description: In this problem you will be given an undirected unweighted graph G. You have to identify if the given graph is bicolorable or not. The rule of coloring this graph is, for each edge connecting two vertices u and v, must have different colors.

In the first line you will be given two integers V and E denoting the number of vertices and edges of this graph respectively. All the vertices will have ids between 1 to V. In the following E lines, you will get the information about all the edges. In the i^{th} line you will have two integers x (1<=x<=V) and y (1<=y<=V) which denotes there is an undirected edge between x and y.

As output, you need to print a string “YES” without quotes if the graph is bi-colorable else print “NO”.

Limits
1<=V<=100000, 1<=|E|<=100000

Code :

```
#include<bits/stdc++.h>
using namespace std;
#define N 100005

vector<int> graph[N];
bool vis[N];
queue<int> q;
int level[N];

bool bfs(int src) {
   vis[src] = true;
   q.push(src);

   bool flag = true;
   while(!q.empty()) {
      int node = q.front();
      q.pop();

      for(auto child: graph[node]) {
         if(vis[child]) {
            if(level[child] != level[node]) {
               continue;
            } else {
               flag = false;
               break;
```

```
            }
        }
        q.push(child);
        vis[child] = true;
        level[child] = level[node]+1;
      }
      if(flag == false)
        break;
   }
   return flag;
}

int main() {
   int V, E;
   cin >> V >> E;
   for(int i = 1; i <= E; i++) {
      int x, y;
      cin >> x >> y;
      graph[x].push_back(y);
      graph[y].push_back(x);
   }
   int res = true;
   for(int i = 1; i <= V; i++) {
      if(vis[i]) {
         continue;
```

```
        }

        res &= bfs(i);

    }

    cout << (res != false ? "YES" : "NO") << endl;



    return 0;

}
```

#Problem 3: Tree Diameter for an undirected tree graph

Description: In this problem you will be given an undirected unweighted Tree graph G. You have to discover the diameter of the given tree. Tree diameter mainly indicates the largest distance found between any two nodes in the given tree. Distance means the number of edges between the nodes.

In the first line you will be given two integers V and E denoting the number of vertices and edges of this graph respectively. All the vertices will have ids between 1 to V. In the following E lines, you will get the information about all the edges. In the i^{th} line you will have two integers x (1<=x<=V) and y (1<=y<=V) which denotes there is an undirected edge between x and y.

You will write a program that will calculate the diameter of the given undirected tree. For each test case you will print a single line denoting the diameter.

Limits
1<=V<=1000, 1<=|E|<=1000000

Code :

```
#include<bits/stdc++.h>

using namespace std;

#define n 1009



vector<int> graph[n];

int vis[n];

int depth[n];
```

```
int dfs(int vertex) {
    static int max = vertex;
    vis[vertex] = true;
    for(auto child: graph[vertex]) {
        if(vis[child]) {
            continue;
        }
        depth[child] = depth[vertex]+1;
        if(depth[child] > depth[max])
            max = child;
        dfs(child);
    }
    return max;
}

int main() {
    int V, E;
    cin >> V >> E;

    for(int i = 1; i <= E; i++) {
        int x, y;
        cin >> x >> y;
        graph[x].push_back(y);
        graph[y].push_back(x);
    }
```

```
    int r1 = dfs(1);

    memset(vis, 0, sizeof(vis));

    memset(depth, 0, sizeof(depth));

    int r2 = dfs(r1);



    cout << depth[r2] << endl;



    return 0;

}
```

#Problem 4: Multiple shortest path existence

Description: In this problem you will be given an undirected unweighted (or each edge equally weighted) graph G and a source vertex S. You need to find if there exists any node X that can be reached in multiple shortest paths from S. In this problem, the shortest path between two vertices means using the minimum number of edges to reach from one vertex to the other.

In the first line you will be given two integers V and E denoting the number of vertices and edges of this graph respectively. All the vertices will have ids between 1 to V. In the following E lines, you will get the information about all the edges. In the i^{th} line you will have two integers x (1<=x<=V) and y (1<=y<=V) which denotes there is an undirected edge between x and y. After E lines of input, you will get two a single integer S denoting the source vertex of the problem.

As output, you will either print “YES” (witout quotes) if there exists at least one node having multiple shortest paths to reach from S or print “NO” (without quotes) if there exists no such node.

Limits
1<=V<=100000, 1<=|E|<=100000

Code :

```
#include<bits/stdc++.h>

using namespace std;

#define N 100005
```

```
vector<int> graph[N];
bool vis[N];
queue<int> q;
int level[N];

void bfs(int src) {
    vis[src] = true;
    int flag = 0;
    q.push(src);

    while(!q.empty()) {
        int node = q.front();
        q.pop();

        for(auto child: graph[node]) {
            if(vis[child]) {
                if(level[child] > level[node]) {
                    flag = 1;
                    break;
                }
                continue;
            }
            q.push(child);
            vis[child] = true;
            level[child] = level[node]+1;
```

```
        }
        if(flag == 1)
            break;
    }
    cout << (flag != 0 ? "YES" : "NO") << endl;
}

int main() {
    int V, E;
    cin >> V >> E;
    for(int i = 1; i <= E; i++) {
        int x, y;
        cin >> x >> y;
        graph[x].push_back(y);
        graph[y].push_back(x);
    }
    int S;
    cin >> S;
    bfs(S);

    return 0;
}
```

Lab Finish.

Project: Sequence Alignment Problem

In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. It is a fundamental problem in Biological Science.

Given as an input two strings, $X = x_1 x_2....x_m$, and $Y = y_1 y_2....y_m$, output the alignment of the strings, character by character, so that the net penalty is minimized. The penalty is calculated as:

1. A penalty of P_{gap} occurs if a gap is inserted between the strings.
2. A penalty of P_{xy} occurs for miss-matching the characters of X and Y.

Code :

```
#include<iostream>

using namespace std;


  int panal[50][50],temp,gap_panal,miss_panal,minV;

   char par[10][10];


int min_panal(int i,int j,int k)

{

   if (i<=j)

   {

      minV=i;

   }

   else

   {

      minV=j;

   }

   if(k<minV)

   {
```

```
        minV=k;
    }
    return minV;
}
int getMatch(char a,char b,int c)
{
    if (a==b)
    {
        return 0;
    }
    else
    {
        return c;
    }
}
void print_alignment_seq1(int panlal[][50],char par[][10],string str1,int i,int j)
{
  if(par[i][j]=='D')
  {
    print_alignment_seq1(panlal,par,str1, i-1, j-1);
    cout<<str1[j-1];

  }
  else if(par[i][j]=='V')
  {
```

```
   print_alignment_seq1(panlal,par,str1, i-1, j);
   cout<<'_';
  }
  else if(par[i][j]=='H')
  {
     print_alignment_seq1(panlal,par,str1, i, j-1);
   cout<<str1[j-1];
  }
}
void print_alignment_seq2(int panlal[][50],char par[][10],string str2,int i,int j)
{
  if(par[i][j]=='D')
  {
   print_alignment_seq2(panlal,par,str2, i-1, j-1);
   cout<<str2[i-1];

  }
  else if(par[i][j]=='V')
  {
   print_alignment_seq2(panlal,par,str2, i-1, j);
   cout<<str2[i-1];
  }
  else if(par[i][j]=='H')
  {
   print_alignment_seq1(panlal,par,str2, i, j-1);
```

```
    cout<<'_';
  }
}
int main()
{
    string str1,str2;

    cout<<"enter your two string"<<endl;
    cin>>str1>>str2;
    cout<<"enter gap panelty"<<endl;
    cin>>gap_panal;
    cout<<"enter missmatch panelty"<<endl;
    cin>>miss_panal;

    for (int i=0; i<str2.length()+1; i++)
    {
        for(int j=0; j<str1.length()+1; j++)
        {
            if(i==0&&j==0)
            {
                panal[i][j]=0;
            }
            else if(i==0)
            {
                panal[i][j]=panal[i][j-1]+gap_panal;
```

```
            par[i][j]='H';
        }
        else if(j==0)
        {
            panal[i][j]=panal[i-1][j]+gap_panal;
            par[i][j]='V';
        }
        else
        {
            panal[i][j]= min_panal(panal[i-1][j-1]+getMatch(str2[i-1],str1[j-1],miss_panal) ,panal[i-
1][j]+gap_panal, panal[i][j-1]+gap_panal);
            temp=panal[i][j];
            if (panal[i-1][j-1]+getMatch(str2[i-1],str1[j-1],miss_panal)==temp)
            {
            par[i][j]='D';

            }
            else if (panal[i-1][j]+gap_panal==temp)
            {
             par[i][j]='V';
            }
            else if (panal[i][j-1]+gap_panal==temp)
            {
              par[i][j]='H';
            }
      }
```

```
    }
    cout<<endl;
}
cout<<"minimum panalty "<<panal[str2.length()][str1.length()]<<endl;

print_alignment_seq1(panal,par,str1,str2.length(),str1.length());
cout<<endl;
print_alignment_seq2(panal,par,str2,str2.length(),str1.length());

}
```

Conclusion : We can solve (Lab code + Project code).